The long long road to pkg_add -u

Marc Espie (espie@openbsd.org)

october 2010, sunday 10

In the beginning (2000 ?)

There were the tools from Jordan Hubbard, with a lot of drawbacks:

- they were slow
- they were in C
- they were a hack

and one good point

- they existed

Who am I

At this point in time,

- I had been involved with OpenBSD ports for about five years.
- I was "chief architect" of the ports tree.
- I had rewritten a lot of the .mk file,
- and I had taken over make itself.

but this is a topic for another talk.

Some design goals

- Focus on binary packages. Only porters should build packages (Theo’s insight)
- Be safe. C is fast and everything, but a summary audit of pkg_add showed tons of possible buffer overflows.
- Having updates would be cool eventually.
- Stop reinventing the wheel. We shouldn’t have tens of scripts that parse package manifests.
- Be fast. Users don’t want to wait for packages.

Bonus goals

- Be compatible with existing stuff
- Text /var/db/pkg is nice

The most controversial decision: Perl

Why not

- A lot of people don’t like perl for irrational reasons
- Write-only code (but see IOCCC)
- It could be slow
- Long start-up time

But

- I like it
- Need a RAD platform, let’s take one I know
- Very modular. Nice namespace system
- Perl is part of the base system in OpenBSD

Architectural and practical goals

Architectural goals

- clean API for package manifests (packing-lists)
- fluid design, it will evolve

Practical goals

- Acceptance of the new tools
- Complete replacement
- Fast enough

Architectural goals

Six months (and several iterations later)

- a design I liked.

In the end

- packing-lists are structured objects
- They can be read/written.
- This validates and normalizes the structure.

Each object is a packing-element (base class) & further properties.

- Differences are implemented as methods
- Perl bonus: classes are "open"
  - you can add a visitor later, as an after-thought

All operations in the ports tree that manipulate packing-lists

use this abstraction.

Architectural goals (cont.)

- Objects are stored in lists according to type.
- Meta-information "migrates" to the top.
- Positional information (such as @mode/@owner) migrates to every object

Goal achieved:

- Clean design that scales well.

That "core" of pkg_add is just glorified MANIFEST handling.

It has to be perfect !

Practical goals

The initial replacement got faster than the C version.

The C version used external tar(1), where the perl version unpacked its archive itself (using its own Ustar module), so no staging area required.

Turned out the most expensive operation was copying files around.

- Dropped-in replacement for pkg_add/pkg_delete in 2003:
  - no showstopper bug!
- Used Ustar write support for pkg_create in 2005

BSD is dying

and SmallTalk is dead.

- (Ask PHK, everyone’s doing Lisp^WXML)
- But you can still write SmallTalk in perl
- open classes can be extended later
- hashes are nice for adding to data structure later
- modules can use the same hash, not knowing about each other
- not many collisions

- Same optimization techniques
- Same drawbacks
- Same benefits

Marketing mistake

In retrospect, keeping the same name was cool internally, and a mistake with respect to other projects.

Newcomers expect the same clunky dumb tools that exist on *BSD, whereas our pkg_add has nothing in common with those.

A design observation

Most package tools are built as "smart" tools that call "dumb" tools.

This is wrong WRONG WRONG !!!

Dumb tools will use only the information they need. Smart tools have to "discover" things: deduce semantic information from what dumb tools tell them, and reparse stuff to get additional info.

Lots of processing power wasted. Redoing the same thing over and over.
Problems in dumb tools are hard to solve, because dumb tools don’t have enough information to take smart action.

A design observation (cont.)

Dumb tools will need automatic generation.

Sometimes this works

- autoconf is a shining example of that

Seriously

- auto-generate scripts = auto-generate BUGS

Smart API: benefits

- All package tools use the same interface to packing-lists
  - all the information is exposed.
  - Tools can grab whatever they want.
- Full semantic information
  - everything relevant to a file/other object is there and can be used.
- During a package addition, there’s a single instance of pkg_add.
  - doesn’t have to rescan /var/db/pkg.
  - just needs to keep its internal information synchronized.
- The same information can be used by validation checkers.
  - same API, exact same semantic info as pkg_add

Smart API benefits: reuse

- 2003: find-all-conflicts
- 2004: check-lib-depends
- 2004: make-plist
- 2005: register-plist

adding stuff to packing-lists is trivial

- you just need to write one method or two once
- don’t repeat yourself EVER

ongoing work

- PackingList/Element API is stable since 2008 ?

pkg_add -u, up in the sky

Impossible design goal

- type pkg_add -u and have it update everything

OpenBSD: realistic goals

Don’t try to do sudoku in pkg_add (any debianists ?)

Need for stepping stones

- how do I update one single package ?
- how will that break if I update more than one ?
- how do I discover what I need to update ?

pkg_add -r, that’s cheating

So the initial idea was to be able to replace one single package.

Happened in 2004-2005.

That’s pkg_add -r:

- you pass it the new package name, and it replaces things.

- Replacement works backwards: you deduce the old name from the new one
- Replacement must be safe.

Transactional vs. provable semantics

A lot of package systems out there do transactional semantics.

- try to update
- if it f* up, go back to previous state.

We do provable replacements

- compute as much as we can to ensure things won’t fail
- once we’re satisfied, do the replacement (that can’t fail)

Works most of the time

- We now have tools (pkg_check, 2010) in the remaining cases.

Provable semantics

- Check dependencies still match
- Verify the file system will fit (vstat)
- Extract all files in temporary locations
- Do various other things

The temporary location is as close as possible to the final one (same directory usually), so if we can write the file, we can move it.

Only case where it fails is catastrophic failure (panic!!!)...
or bugs in pkg_add (shit happens)

Visitor, again

For instance:

- package addition is a module Add.pm
  - visits a packing-list, calling install on each object
- for replacement
  - visit old list for validation
  - visit new list for validation
  - visit new list with extract (temporary file)
  - visit old list with delete
  - visit new list again with install

Matching dependencies

Both forward, and backwards.

- to install a new package, dependencies must already be there
- to replace a package, stuff that depends on it must still work
- libraries are a problem

Solving the library problem, ports

Developers upstream don’t understand ABI issues.

They’re too busy converting to XML...

The system must take control: change typedef size_t, and have all

C++ libraries be incompatible.

- long and painful process: we control every shared library
- lots of people helped
- (there’s some magic for libtool and cmake and...)

Solving the library problem, packages

Package dependencies: do libraries independently.

- A package that wants a given library has a @wantlib in its packing-list.
- This @wantlib is inserted very late
- and dependent on the current system.

Packages register their libraries: those files are tagged with @lib.

A library will be found

- if there’s a @lib that matches a @wantlib somewhere
- in the @depend tree during installation
- or in the base system

Solving the library problem, updates

- Ties between @lib and @wantlib are stored under /var/db/pkg.

During an update,

- old libraries are kept and put in stub packages.
- They’re only replaced if the ABI is the same.

The stub packages can be removed

- once all dependent packages have been updated.

Maximal reuse:

- stub packages are normal packages

Working replace

In 2005 pkg_add -r did start working.

OpenBSD was able to update packages by specifying a list of new packages

- Replacing one package at a time
- Start on the inside (packages with no dependencies)
- End on the outside (packages with all dependencies)
- Safe: each individual replacement was checked before performing it.

Speaking of the devil

(Hi, Theo):

Details, details, details

- fonts are special
- libraries require ldconfig
- info files are weird
- directories can be shared
- when do we create new users

one single pkg_add running

- common data structures
- stash structured hashes
- use data when needed (visitor pattern)

For instance

the old pkg_add required @exec ldconfig annotations.

the new one knows about @lib, and @exec, and runs ldconfig just in time.

Thus being much faster.

@dirrm is gone. Directories are handled as shared items (last package out removes the directory)

pkg_add -u, cheating version

Running pkg_add -r is tedious: you must know all package names.

Let’s discover them instead (Aug. 2005).

- We have clean package names: stem-version-flavor
- To update, look at packages that share the same stem
- Keep only the packages that conflict
- Keep only packages coming from the same ports directory

pkg_add -u, cheating version (cont.)

For instance,

- to update mutt-1.4,
- mutt-1.5 and mutt-1.4.1 are candidates
- they conflict with mutt-1.4 (@conflict mutt-*)
- mutt-1.4 came from mail/mutt/stable
- mutt-1.4.1 comes from mail/mutt/stable
- mutt-1.5 comes from mail/mutt/snapshot
  - choose mutt-1.4.1

Look ma, no database

Half a design goal was to keep things dead simple: we stored text files and under /var/db/pkg, and we cache absolutely nothing.

As an OpenBSD developer, I’m totally paranoid. cache synchronization does fuck up. If I can get one less failure point, I want to!

So we get update information on the go: open package, scan beginning of packing-list, close package.

It was a game: how far can we get with no db.

Look ma, no database

Turns out we could go ALL THE WAY.

We still do not have any database.

Big toll on ftp (lots of open/close connections).

We have plans for http.

Good design:

- forces sensible package names.
- pkg_add can deduce most things from package names,
- and so can the user.
- There are few exceptions.

But it’s cheating!

- Notice we don’t use version numbers
  - This can downgrade packages
  - Okay it won’t, since OpenBSD has complete snapshots
- We don’t deal with dependencies problems.
  - If two packages are tied (say pgsql-client/server), we update one, then the other. Even though the system says no.

Slow going

- 1/ discover all updates
- 2/ run each of them as a replacement

If something breaks, you’re back to 1/. Finding updates is slow.

Where do we go from there (Once more with feeling)

Plan to do better updates.

- incremental stuff, so we update as we find them
- actually use version numbers.

Details again

- a lot of special cases showed up
- most of them were difficult to predict

Good plan

- impossible to design for everything from scratch
- get it 99% of the way working, then solve the 1%.
- we can’t predict the future
- perl is good: fluidity

Weird shit, good shitz

- files move between packages
- dependency inversions happen
- tied updates should be handled
- packages will get renamed, or disappear
- version numbers should be handled

UpdateSets (2007-2010)

We model a full update as a set of small atomic operations.

Replacements were old package -> new package.

An UpdateSet is (set of old) -> set of new.

As small as possible, so if an update stops, your system still works.

Tracker and UpdateSets

- pkg_add creates a list of UpdateSets
- some module is responsible for filling the blanks
- the engine checks that an UpdateSet is complete
- if it’s not, the engine merges the UpdateSet with what’s needed.

- Tracker is responsible for all UpdateSets
- The replacement engine is responsible for merging stuff
- The dependency engine cooperates with the Tracker to process UpdateSets in the right order.

In theory...

all is good.

- We discover updates on the fly
- pkg_add starts working right away
- UpdateSets are very small
- as safe as possible

In practice...

- very complicated: quite a few bugs
- some updatesets are less small than others.
- very slow

libfam -> avahi update triggerred "big" updatesets: 50 packages to update in one go.

pkg_add would take >1 minute

for one iteration of the tracker engine.

Publicity

But perl is very good. It has a killer profiler. If you use perl, use

NYTProf

best profiler ever.

After better caching and optimizing (normal Smalltalk tricks),

- pkg_add was back to instantaneous for this,
- and faster for normal cases.

Sugar and details

pkg_add has a progress bar.

pkg_add has quirks: quirks is a specific package that contains all exceptions to naming problems.

So we handle:

- renames
- stuff in the base system

okay, database... if you can call a database a list of ~30 package names.

Big detail: signatures

When do we update a package ?

- when it changes
- ... or when its build dependencies change
- so each package records its dependencies: @depend and @wantlib
- that’s a package signature

Confusing:

- internally, pkg_add manipulates package locations
  - they have names
  - they come from somewhere
  - two packages of the same names can be different

Looking back

The main mistake I did was not look at version numbers earlier.
Cheating on pkg_add -r was very costly.

People didn’t get the rules for pkg_add -u.

After adding a lot of error messages to pkg_add, and fixing problems, we have clean stuff now.

It is still complicated, but it is solving a complicated problem !

Current pkg_add (and tools) is 15000 lines of perl.

Looking back (cont.)

One mistake I did not make was try to solve it at once

- pkg_add -u is a practical tool
- initial design goals were quickly met

But you can’t predict the future

- ran into unexpected problems
- ran into inefficiencies
- have very hard-to-please users...

OpenBSD is an hostile environment, and that’s GOOD for quality.

Keeping up with the Jones

I keep a close look on apt, pkgsrc, rpm, pkg_upgrade...

- we’re better than all of them
- ... because we have our design goals
- stability and reproducability
- less knobs
- same principle as the rest of OpenBSD
- including OpenSSH

The future

Currently pkg_add is fastest using scp: it uses the rsync trick.

http 1.1 would make things faster.

- It supports byte-range
- so we can "guess" at what we need from a packing-list, and bring a package in slowly.

pkgin frontend

- We don’t need pkgin. Our pkg_add does everything pkgin does.
- But the pkgin UI is nice. It’s just a question of writing it.

more ldconfig sugar

- write a packing-list interface to common operations,
- stuff like @update-desktop-database doesn’t run 20 times during a gnome update.

Thank you

- to my fellow porters
- to my fellow users
- to my audience

- Any questions ?